Konzepte der persistenten Programmiersprache PLEX

Dipl.-Inf. Jürgen Schlegelmilch *

16. Oktober 1996

Zusammenfassung

Zu einem objektorientierten Datenbanksystem (OODBMS) gehört eine Programmiersprache zur Methoden-Implementierung und Anwendungsentwicklung. Existierende objektorientierte Programmiersprachen sind dafür nicht geeignet, da sie zum einen Persistenz, zum anderen das Datenmodell des Datenbanksystems nicht berücksichtigen. Neben diesen Datenbank-spezifischen Anforderungen haben auch existierende objektorientierte Programmiersprachen noch einige Schwachpunkte, die bei einer Neu-Konzeption vermieden werden können. Für das OODBMS OSCAR wird im Moment die Programmiersprache PLEX konzipiert, aus der Menge der Konzepte sollen hier zwei vorgestellt werden: Migratoren und Signatur-Varianz.

1 Einführung

1.1 Motivation

Objektorientierte Datenbanksysteme fassen Daten und darauf arbeitende Funktionen zu Objekten zusammen. Für die Beschreibung der Daten wird ein Datenmodell verwendet, während die Implementierung der Funktionen — nun Methoden genannt — in einer Programmiersprache geschieht. Sowohl die Daten als auch die Methoden werden in einer Datenbank abgelegt (persistent gemacht) und existieren unabhängig von Anwendungen. An Programmiersprachen für Datenbanksystem werden besondere Forderungen gestellt, unter anderem:

Adäquatheit: Die Programmiersprache soll alle Möglichkeiten des Datenmodells ausschöpfen, um eine angemessene Behandlung der Objektdaten zu ermöglichen.

Sicherheit: In Datenbanken ist die sichere Ausführung von Operationen wichtig; dies muß auch für Methoden gelten. Die Programmiersprache muß die sichere Programmentwicklung und -ausführung und das Transaktionskonzept unterstützen.

Effizienz: Datenbankanwendungen arbeiten typischerweise auf sehr großen Datenmengen, Methoden werden für große Objektmengen ausgeführt. Die Konstrukte der Datenbankprogrammiersprache sollten also effizient ausführbar sein.

Für herkömmliche objektorientierte Programmiersprachen stellen sich viele der Forderungen nicht, da diese nur mit ihrem eigenen Datenmodell arbeiten müssen, nur vergleichsweise kleine Datenmengen behandelt werden und keine Daten die Laufzeit eines Programmes überdauern.

*FB Informatik, Uni Rostock, 18051 Rostock; email: schlegel@informatik.uni-rostock.de

1.2 Datenbankprogrammierung mit OSCAR

Das OODBMS OSCAR ([Kla92]) entstand rund um das Datenmodell EXTREM mit der dazugehörigen Objektalgebra ABRAXAS ([HHRW93]) und wurde zunächst ohne Operationenteil entworfen und implementiert. Nachdem zunächst der Bereich der Anfragesprachen ausgebaut wurde, ist die Konzeption einer Datenbankprogrammiersprache langsamer vorangeschritten ([Wei91], [Sch92]) und wird in [Coe94] einen vorläufigen Abschluß erreichen.
Während andere OODBMS sich direkt an eine Programmiersprache anlehnen und im wesentlichen deren Datenmodell übernehmen, ist das Datenmodell von OSCAR eine eigenständige Entwicklung: EXTREM umfaßt unter anderem folgende Besonderheiten:

- Werte und Objekte sind strikt getrennt.
- Werte werden durch ADTs beschrieben, Objekte durch Klassen.
- EXTREM unterstützt Mehrfachvererbung zwischen Klassen.
- Ein Objekt kann zu vielen, auch unvergleichbaren Klassen gehören.
- Die Klassenzugehörigkeit eines Objekts kann wechseln.

Insbesondere die letzten beiden Punkte sind in anderen objektorientierten Systemen sehr selten zu finden. Deshalb mußte eine eigene Programmiersprache entwickelt werden, die auf dem Datenmodell EXTREM aufsetzt und die oben genannten Anforderungen berücksichtigt.

2 PLEX — die Sprache

Im Rahmen einer Diplomarbeit ([Coe94]) am Insitut für Informatik der TU Clausthal werden derzeit die bestehenden Konzepte für eine persistente Programmiersprache für EXTREM zusammengestellt und erweitert; als Grundlage dient dabei die Sprache Eiffel ([Mey93]). PLEX (Programming Language for EXtrem) ist also eine objektorientierte, blockstrukturierte Sprache mit folgenden Charakteristika:

- EXTREM als Datenmodell wird in vollem Umfang unterstützt.
- Klassen und ADTs beschreiben gültige Element-Zustände mit Invarianten.
- Der dynamische Typ von Objekten kann zur Laufzeit festgestellt werden.
- Für transiente Objekte gibt es eine Freispeicherverwaltung.
- Methoden haben Vor- und Nachbedingungen und eine Ausnahmebehandlung; bei Redefinition gilt die Covarianz-Regel.
- Migratoren und Demigratoren verschieben Objekte in und aus Klassen.

In diesem Artikel werden zwei der Konzepte vorgestellt, und zwar die Regel für Signatur-Varianz bei Methoden-Redefinition und das Konzept der Migratoren und Demigratoren.

2.1 Varianz der Signatur bei Redefinition

2.1.1 Einleitung

Methoden werden von Oberklassen an Unterklassen vererbt und können dort durch Redefinition mit einer neuen Implementierung versehen werden. Dabei möchte man gerne die Signatur der Methode variieren, da die speziellere Version oft auch speziellere Argumente erfordert. Ein Beispiel zeigt diese Situation:

> Eine Klasse `LinkList` für einfach verkettete Listenelemente hat die Methode `add(aSuccesor:LinkList)` zum Aufbau von Listen. Davon wird die Klasse `DblLinkList` für Elemente mit doppelter Verkettung abgeleitet, die Methode `add` muß dazu neu implementiert werden. Die naheliegende Redefinition ist `add(aSuccessor:DblLinkList)`.

Diese Art der Varianz der Signatur wird Covarianz genannt; Eiffel [Mey93] und O_2 [Deu91] verwenden sie. Sie ist jedoch nicht typsicher, da sie gegen das Substituierbarkeitsprinzip verstößt:

> In einer Methode sei der Parameter `list` mit dem Typ `LinkList` deklariert; mit `list.add(item)` kann man das `LinkList`-Objekt `item` an `list` anhängen. Durch Substituierbarkeit kann `list` aber ein `DblLinkList`-Objekt enthalten, Late binding ruft dann `DblLinkList.add(aSuccessor:DblLinkList)` auf — mit dem `LinkList`-Objekt `item` als Argument.

2.1.2 Lösungsansätze

Sei $\prec$ die Unterklassen-Beziehung, $B \prec A$ zwei Klassen und M_A, M_B eine Methode in A bzw. ihre Redefinition in B. Die Signaturen der Methoden seien $M_A : C_1 \times \cdots \times C_n \to C_0$ und $M_B : D_1 \times \cdots \times D_n \to D_0$. Dann definiert man:

$$\begin{array}{lcl} M_B \text{ redefiniert } M_A \text{ covariant} & \iff & D_i \preceq C_i, i \in [0:n] \\ M_B \text{ redefiniert } M_A \text{ contravariant} & \iff & C_i \preceq D_i, i \in [1:n], D_0 \preceq C_0 \end{array}$$

Contravarianz ist immer typsicher, sie wird etwa in Oberon [RW92] verwendet. Covarianz ist so nicht typsicher; ein erweiterter Typcheck wurde für Eiffel in [Mey93] vorgestellt, der durch Datenflußanalyse die Typfehler zu erkennen versucht. C++ [Str91] erlaubt keine Signatur-Varianz bei Redefinition (Novarianz) und verwendet statt dessen Overloading, wo die statische Typinformation zum Binden dient.
Eine völlig andere Lösung sind Multimethoden (etwa in Cecil [Cha93]), wo der dynamische Typ aller beteiligten Objekte zum Binden verwendet wird (multi-dispatched):

> Als Multimethode wird `add` mit `add(prev:LinkList,next:LinkList)` und `add(prev:DblLinkList,next:DblLinkList)` definiert. Im Falle der gemischten Argumenttypen wird die erste Implementierung genommen, man erhält also eine einfach verkettete Liste.

Multimethoden sind typsicher schon mit üblichen Typchecks, erfordern aber höheren Aufwand beim Binden und passen nicht in ein Datenmodell, das nur Klassen kennt.

2.1.3 Die PLEX-Lösung

PLEX verwendet die Covarianz-Regel, jedoch nur einen herkömmlichen Typcheck. Damit kann es zu Typfehlern während der Laufzeit kommen, die aber von den Ausnahmebehandlungsmöglichkeiten in PLEX abgefangen werden. Ein Datenbanksystem muß ohnehin auf Ausnahmesituationen sinnvoll reagieren können; in PLEX hat darum jede Methode einen Ausnahmebehandlungsteil, in dem individuell auf jede Ausnahme reagiert werden kann. Der Aufruf einer covarianten Redefinition mit falschen Parameter-Typen löst eine Ausnahme aus, sodaß die aufrufende Routine entsprechend reagieren kann.
Diese Lösung kann nicht voll befriedigen. Das nächste Ziel wird darum sein, das Datenmodell zu erweitern, um eine saubere Integration von Multimethoden zu ermöglichen, die den Aufwand für Multi-Dispatching minimiert.

2.2 Migratoren und Demigratoren

Das Datenmodell EXTREM ermöglicht einem Objekt den Wechsel der Klassenzugehörigkeit in weiten Grenzen (vgl. [HH91]). Dies wird in PLEX durch die Verallgemeinerung der Konstruktoren und Dekonstruktoren zu Migratoren und Demigratoren realisiert; beides sind spezielle Updatemethoden.

2.2.1 Migratoren

Ein Migrator M nimmt ein Objekt einer Klasse C und fügt es in eine andere Klasse D ein; C heißt Quellklasse, D die Zielklasse des Migrators. Die Hauptaufgabe dabei ist die Initialisierung neuer und die Anpassung vorhandener Attribute, um die Klasseninvariante von D zu erfüllen. Um auf Objekte aus C anwendbar zu sein, muß M in C deklariert sein; um Attribute von D initialisieren zu können, muß er in D definiert sein. Migratoren nehmen also eine Sonderstellung ein; syntaktisch werden sie in PLEX in der Zielklasse deklariert und definiert. M wird an Unterklassen von C vererbt wie andere Methoden, jedoch ist seine Anwendung auf Objekte aus D und Unterklassen meist unerwünscht und sollte in der Vorbedingung ausgeschlossen werden.
M kann ein Objekt erst in D einfügen, wenn es in allen Oberklassen von D ist; dies muß in der Vorbedingung berücksichtigt werden. Der Migrator kann also davon ausgehen, daß die Attribute aller Oberklassen zur Initialisierung der neuen zur Verfügung stehen; dazu kommen eventuelle Parameter des Migrators. Eine Migration aus den direkten Oberklassen wird durch sogenannte Einfachmigratoren (kurz: Migratoren) erledigt. Diese Art der Migratoren haben die Menge der direkten Oberklassen ihrer Zielklasse als Quellklasse. Für eine abstrakte Klasse als Zielklasse ist der Migrator gleichzeitig der Konstruktor.
Die Migration über mehrere Spezialisierungskanten hinweg wird von Fernmigratoren übernommen. Sie haben genau eine Quellklasse und müssen die Gesamtmigration in eine Folge von Einfachmigrationen zerlegen. Da Migratoren geerbte Attribute ändern dürfen, damit die Invariante der Zielklasse erfüllt wird, spielt die Reihenfolge der Einfachmigrationen eine Rolle, zudem müssen die Migratoren mit passenden Argumenten aufgerufen werden. Eine automatische Fernmigration ist deshalb nicht möglich.
Probleme bereiten Generalisierungsklassen, da Objekte automatisch in sie migrieren. Darum müssen die Migratoren aller Unterklassen einer Generalisierung C prüfen, ob das Objekt schon in C ist, und andernfalls den Migrator von C aufrufen. Hier ähnelt die Einfachmigration in eine Unterklasse von C der Fernmigration, da sie weitere Migrationsschritte erzwingt.

Bei Fernmigration aus Klasse Q in Klasse Z ist nicht von vornherein klar, in welchen Klassen das Objekt schon ist und in welche es migriert werden muß. Dazu bildet man die Menge $K_Q = \{C \in \mathit{extent}(\mathsf{Class}) | \, Q \preceq C\}$ der Klassen, in denen das Objekt schon ist, und die Menge $K_Z = \{C \in \mathit{extent}(\mathsf{Class}) | \, Z \preceq C\}$ derer, in denen es hinterher sein muß. Die Menge K_M der Klassen, in die es eventuell erst hineinmigrieren muß, ergibt sich dann als Schnitt: $K_M = K_Z \setminus K_Q$. Aus K_M kann man dann alle Generalisierungsklassen entfernen, da die Einfachmigratoren der Unterklassen die notwendigen Migratoren aufrufen. Der Programmierer muß die Ergebnismenge nun topologisch sortieren und für jede Klasse C darin eine Anweisung der Form

`if not(o in` C`.extent) then o.make_`C`(...) endif`

schreiben, mit dem Einfachmigrator `make_`C[1].

2.2.2 Demigratoren

Demigratoren sind problemloser als Migratoren: Sie dürfen keine Parameter haben und werden bei Bedarf auch automatisch aufgerufen. Ihre Aufgabe ist die Kontrolle der Demigration durch ihre Vorbedingung und eventuelle Protokollierungsarbeiten. Für abstrakte Klassen entspricht der Demigrator dem Destruktor normaler Programmiersprachen; er hat wegen der Freispeicherverwaltung bzw. Persistenz allerdings nicht dessen typische Aufgabe des Aufräumens.

Literatur

[Cha93] Craig Chambers. The Cecil language: Specification and Rationale. Technical report, University of Washington, Department of computer science and Engineering, FR-35, University of Washington, Seattle, Washington 98195 USA, March 1993.

[Coe94] Ralf Coenning. Konzeption einer persistenten Datenbankprogrammiersprache für OSCAR. Diplomarbeit, TU Clausthal, Institut für Informatik, 1994.

[Deu91] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[HH91] C. Hörner and A. Heuer. EXTREM — The structural part of an object-oriented database model. Informatik-Bericht 91/5, TU Clausthal, October 1991.

[HHRW93] A. Heuer, C. Hörner, H. Riedel, and U. Wiebking. Syntax, semantics, and evaluation of the EXTREM object algebra. Informatik-Bericht, TU Clausthal, 1993. In Vorbereitung.

[Kla92] T. Klaustal. Das OSCAR-Projekt. Informatik-Bericht 92/2, TU Clausthal, 1992.

[Mey93] Bertrand Meyer. *Eiffel: The Language*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, 1993.

[RW92] Michael Reiser and Niklaus Wirth. *Programming in OBERON — steps beyond Pascal and Modula*. acm press. Addison-Wesley, New York, 1992.

[Sch92] J. Schlegelmilch. Vererbung von Methoden in OSCAR. Master's thesis, Institut für Informatik, TU Clausthal, October 1992.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2 edition, 1991.

[Wei91] T. Weik. Methoden in OSCAR. Master's thesis, TU Clausthal, Institut für Informatik, 1991.

[1] Es sind beliebige Namen für Migratoren möglich